

# Design to Win with Integrated FPGA and Microprocessor Solutions

Using “software-compiled system design” for programmable systems, we show how you can combine software and hardware design methodologies — and development tools — from system-level specification to direct implementation and run-time reconfiguration.

by Chris Sullivan  
Director of Strategic Alliances  
Celoxica, Ltd.  
[chris.sullivan@celoxica.com](mailto:chris.sullivan@celoxica.com)

As FPGAs have developed from logic prototyping devices into fundamental system elements, there has been enthusiasm for the concept of using high-performance processors closely coupled to or immersed inside the FPGA fabric for applications that require unrivalled levels of performance and flexibility.

In this architecture, the microprocessor typically runs system applications while the FPGA manages computationally intensive tasks. Offloading processor-intensive tasks to hardware reduces the load on the processor and delivers greater bandwidth. It can also remove bottlenecks by migrating algorithms to hardware. In short, FPGAs have evolved into fully programmable systems and fast co-processors, rather than just flexible relations of the ASIC.

Existing design examples that combine Xilinx FPGA solutions with development tools from Celoxica and Wind River Systems already provide unique and tangible proof that this concept and design flow works. They form a core element of programmable system co-design, delivering a quick, efficient, and verifiable route to device-optimized implementation.

“Software-compiled system design” provides the capability to drive partitioning, co-verification, and direct implementation from the system specification. Moreover, it allows engineers to jump-start their system and software application development before actual hardware is available, thereby enabling concurrent design, saving valuable development effort and delivering the best

time to market. Starting at the system level, verification becomes a whole-design life cycle activity, and by enabling system-level partitioning, you can realize a better quality of design (QoD) — right the first time, more of the time.

## A Design Example

An early design example — developed by Celoxica, Wind River, and Xilinx — focused on the design methodology, tools, and run-time environments that can be applied to programmable systems. Specifically, we developed a triple-DES encryption and decryption engine to compare a programmable system solution with an alternative software implementation. A compressed video stream formed the basis of test data.



Figure 1 - PPMC750 and RC1000 connected and functioning as a prototyping platform for programmable systems

### Hardware

The selected hardware architecture was initially based around a discrete IBM PowerPC™ processor and a Xilinx Virtex™ FPGA – effectively a first-generation Virtex-II Pro™ prototyping platform (Figure 1). We used a PPMC750 single-board computer from Wind River and Celoxica's RC1000 – a Virtex-based PCI card with a Xilinx FPGA and 8 Mb of local memory (Figure 2).

Subsequently, we deployed a newer reference platform using the PowerPC 405GP processor (Figure 3). In addition to PCI and PMC (PCI mezzanine card) connectors, this platform also featured a custom connector that allowed an FPGA daughtercard to be plugged directly onto the processor peripheral bus, thus providing even closer coupling, lower latency, and higher throughput.

Various FPGA daughtercards can be used with this reference platform, such as the ADM-XRC from Alpha Data Systems, Xilinx Durango, or the Proteus card from Wind River.

Wind River's Proteus card is equipped with a Xilinx Virtex device and memory includes 4 MB on-board SSRAM. The FPGA PMC can interface with any standard PMC slot (with an image containing a PCI soft core) or the microprocessor local bus on Wind River's SBC405GP single board computer. There is a substantial performance boost from direct processor bus connection, compared with PCI.

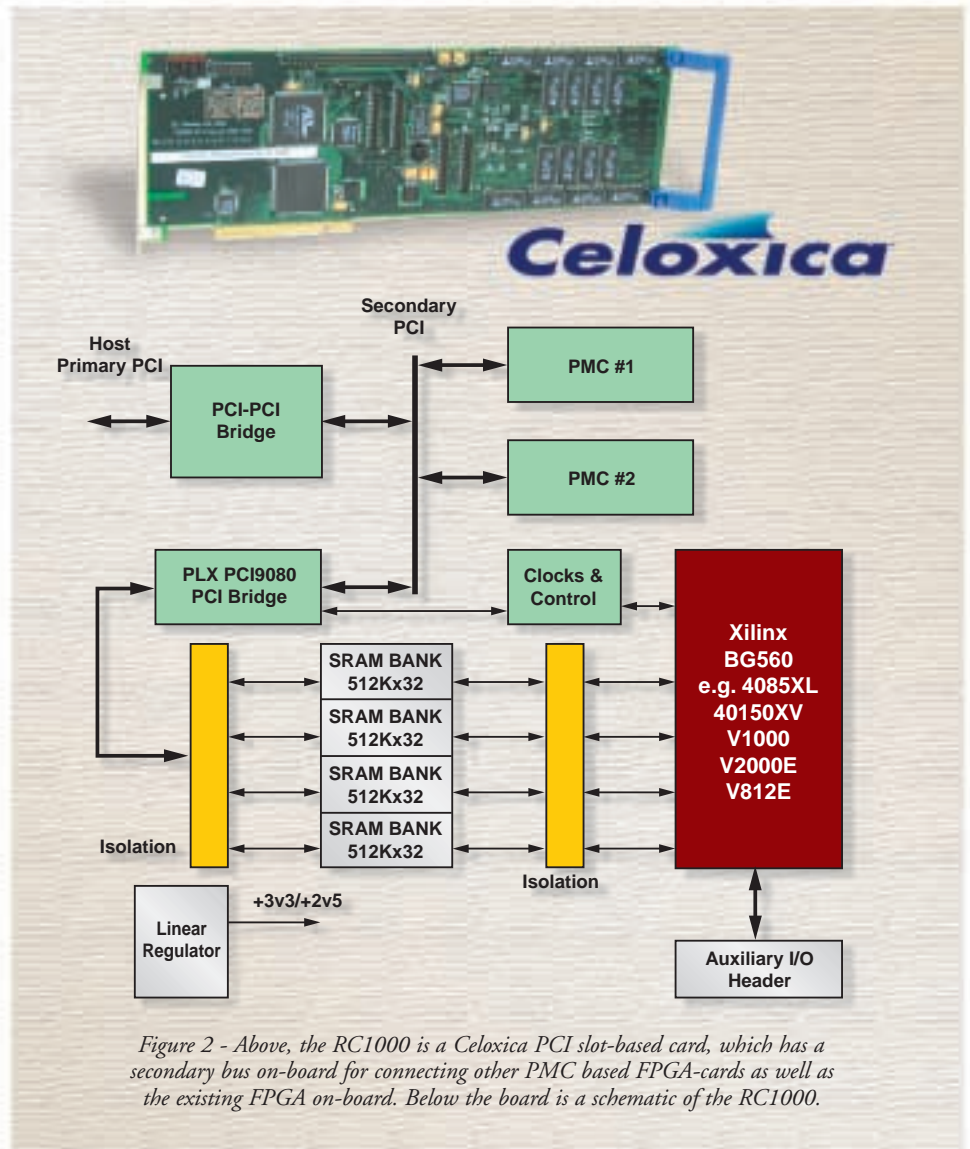


Figure 2 - Above, the RC1000 is a Celoxica PCI slot-based card, which has a secondary bus on-board for connecting other PMC based FPGA-cards as well as the existing FPGA on-board. Below the board is a schematic of the RC1000.

The design platform is completed by a simple DAC interface, enabling the FPGA card to drive a video monitor or a flat-panel LCD for standalone demonstrations.

### Development Tools

The 405GP processor runs Wind River's VxWorks™ real-time operating system (RTOS), together with hardware bring-up tools that allow close control of the boot cycle for the board during the time period before control is passed to the RTOS.

The PAVE Framework API from Xilinx was used to program the FPGA with configuration files.

Determination of the system partition and application content for the FPGA were developed using Celoxica's Nexus co-design environment and DK Design Suite.

### Nexus and DK

Nexus is a powerful co-design environment for programmable systems. It supports system partitioning, co-verification, and co-simulation. Nexus allows you to fully

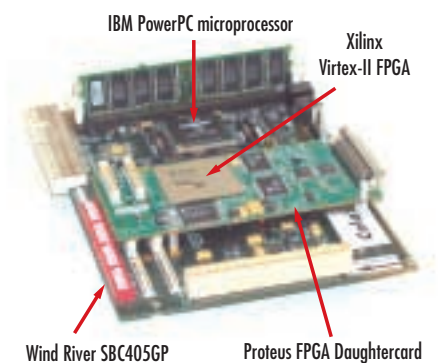


Figure 3 - Virtex-II prototyping platform

explore the design space to identify optimal system partitioning. System functionality can be simulated between hardware and software using multiple languages such as C, C++, Handel-C, SystemC, and HDLs. These models can be used throughout the design for co-verification. Nexus communicates directly during simulation with popular, third-party, hardware RTL simulators and software ISS environments.

Using DK, the resulting code may be debugged using a familiar integrated development environment (IDE), and applications are compiled direct to the FPGA fabric using device-optimized synthesis. VHDL and Verilog output is also supported for traditional RTL synthesis.

### Handel-C

We selected the Handel-C language for hardware implementation, as it provides a common level of abstraction and a common language base for both the hardware and software. The language has simple extensions to ANSI-C (Figure 4) that can be leveraged to quickly create applications that fully exploit the capabilities of a programmable system, without compromising performance or area.

As a fully synthesizable language, everything that can be described in Handel-C has translation to hardware (Figure 5). The code illustrates concepts and extensions, such as `par`, `chan`, synchronization, functions, pointers, structures, interfacing, and externing pure C functions for simulation.

With a simple timing model, each assignment in a program takes one clock cycle to execute, giving you full control over what is happening in the design at any point in time. Results are predictable and controllable, and the facility for complex sequential control flows means there are no state machines to design.

### Run-Time Environment

Typically, the FPGA is connected to the microprocessor in a memory-mapped or programmed I/O fashion, but this creates the challenge of needing to develop and

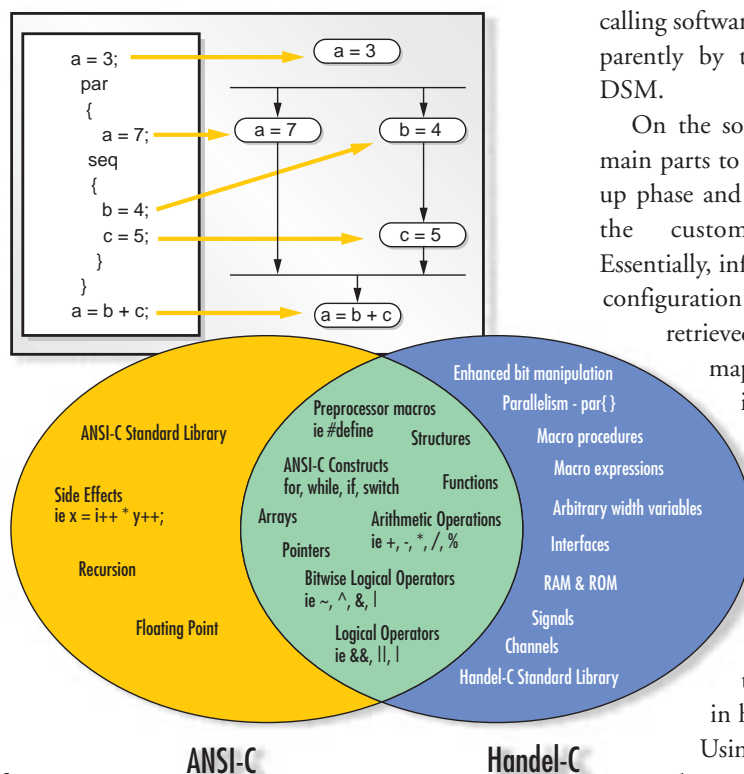


Figure 4 - ANSI-C/Handel-C comparison

redevelop individual communications protocols and data-marshalling routines for each application. This problem is overcome by using DSM (Data Streaming Manager), a portable co-design API developed for hardware/software integration in programmable systems.

### Data Streaming Manager

DSM is a portable hardware/software co-design API that offers a simple and transparent interface for transferring multiple independent streams of data between hardware and software. DSM supports system partitioning and final implementation; it is both bus/interconnect and OS-independent; and for the developer, it simplifies the integration between the hardware and software (Figure 6).

As an example, the hardware function reads parameters from an input port and then writes the results to an output port. All the complexities of receiving commands over the PCI or bus, routing parameters to the appropriate hardware function, and then routing the responses back to the

calling software thread are handled transparently by the hardware side of the DSM.

On the software side, there are two main parts to the DSM: the control/setup phase and then the specific usage of the custom hardware function. Essentially, information about the FPGA configuration and available functions are retrieved by reading a memory-mapped register. User-defined identifiers (called function addresses) are assigned to each available hardware function, and these function addresses are later used to communicate between the application software and the functions implemented in hardware.

Using this methodology, the optimal system partition can be identified by porting blocks of software to Handel-C, for hardware prototyping, testing, and verification. DSM's portability means that multiple partitions can be rapidly evaluated, tested, and verified with the software used as a testbench throughout.

DSM also provides a functionally accurate simulation environment that allows ANSI-C programs and Handel-C applications to interact using the DSM (Figure 7). The ANSI-C program is run as a native executable on the PC. The Handel-C application is run using the simulation and debugging capability of Celoxica's co-design environment. A utility is provided through which the data passing between the applications may be monitored to assist with debugging (Figure 8). All of the API functions are provided, allowing complete system development to begin – without the development platform being available. Once working, the application can be easily transferred to the target platform for final testing.

### Triple DES Encryption

Our design example was based around streaming of compressed and encrypted video data. The Autodesk FLI file format was used to compress the video, and an FLI player, developed by Celoxica, was imple-

mented in FPGA hardware connected to the processor via the PCI bus. To benchmark the design, we loaded a cartoon animation into the memory on the processor board. A triple DES algorithm described in C ran on the PowerPC microprocessor.

The same C source code was ported to Handel-C, optimized in terms of controlling parallelism and timing, and compiled to a gate-level design that was device-optimized for the target FPGA.

A 64-bit key was used for the encryption,

which subsequently allowed correct decryption of the video stream. Implementing three DES algorithms in sequence (triple DES encryption) provided further increases in this standard's security. Three 64-bit keys were used for an encrypt/decrypt/encrypt cycle in a triple DES pass, and the same keys allowed decrypt/encrypt/decrypt for decrypting the data.

This was a robust test of performance. The algorithm was inherently sequential in software, but it could be heavily pipelined for a hardware implementation.

To measure the performance improvement, we played a cartoon with each compressed frame being encrypted, decrypted, and displayed on a VGA monitor. Both hardware and software implementations were displayed together. They were triggered to start simultaneously, with the hardware version programmed to cycle continuously until the software implementation finished. Processed data from the microprocessor was fed to the FPGA, which as well as performing DES encryption

```

define WIDTH 9 // parameterisable data widths
typedef struct // complex number type
{
    signed WIDTH re;
    signed WIDTH im;
}
complex;

set_clock = external " ClockSource ";
void main() // single synchronous clock domain (ClockSource)
{
    chan complex cDataIn[ 2 ], cDataOut; // communication channels

    while(1)
    {
        par // parallel hardware
        {
            DataIO(cDataIn, &cDataOut); // data input/output function
            Transform(cDataIn, &cDataOut); // data transform function
        }
    }

    void Transform(chan complex *pcDataIn, chan <complex> *pcDataOut)
    {
        complex DataInReg[ 2 ], DataOutReg; // complex data registers

        par (i=0; i<2; i++) // replicated par{}
        {
            pcDataIn[ i ] ? DataInReg[ i ]; // read complex numbers in parallel
        }
        par // single cycle multiplication of two complex numbers
        {
            DataOutReg.re = DataInReg[ 0 ].re * DataInReg[ 1 ].re - DataInReg[ 0 ].im * DataInReg[ 1 ].im;
            DataOutReg.im = DataInReg[ 0 ].re * DataInReg[ 1 ].im + DataInReg[ 0 ].im * DataInReg[ 1 ].re;
        }
        *pcDataOut ! DataOutReg; // write complex number
    }

    void DataIO(chan complex *pcDataIn, chan <complex> *pcDataOut)
    {
        complex DataInReg[ 2 ], DataOutReg; // complex data registers

        DataInput( &DataInReg[ 0 ] ); // data input function
        DataInput( &DataInReg[ 1 ] ); // data input function
        par (i=0; i<2; i++) // replicated par{}
        {
            pcDataIn[ i ] ! DataInReg[ i ]; // write complex numbers in parallel
        }
        *pcDataOut ? DataOutReg; // read complex result
        DataOutput( &DataOutReg ); // data output function
    }

    #ifndef SIMULATE // Simulation Testbenches
    void DataInput( complex *pDataIn )
    {
        long Data[ 2 ];
        scanf( "%d%d", &Data[ 0 ], &Data[ 1 ] ); // ANSI-C function
        pDataIn->re = adjs( Data[ 0 ], WIDTH ); // type conversion from ANSI-C call
        pDataIn->im = adjs( Data[ 1 ], WIDTH ); // type conversion from ANSI-C call
    }

    void DataOutput( complex *pDataOut )
    {
        long Data[ 2 ];
        Data[ 0 ] = adjs( pDataOut->re, 32 ); // type conversion for ANSI-C call
        Data[ 1 ] = adjs( pDataOut->im, 32 ); // type conversion for ANSI-C call
        printf( "DataOut: %d %d\n", Data[ 0 ], Data[ 1 ] ); // ANSI-C function
    }

    #else // Hardware Implementations
    void DataInput( complex *pDataIn )
    {
        interface port_in( signed WIDTH in ) DataInPort(); // WIDTH-bit input port
        pDataIn->re = DataInPort.in;
        pDataIn->im = DataInPort.in;
    }

    void DataOutput( complex *pDataOut )
    {
        signed WIDTH Data;
        interface port_out( DataOutPort( signed WIDTH OutPort = Data ); // WIDTH-bit output port
        Data = pDataOut->re;
        Data = pDataOut->im;
    }
    #endif

```

Figure 5 - Handel-C code for single cycle multiplication of two complex numbers

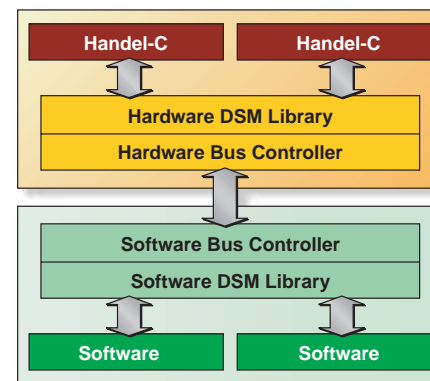


Figure 6 - DSM Data Streaming Manager

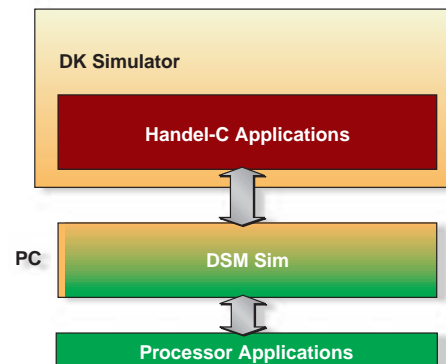


Figure 7 - DSM simulation environment

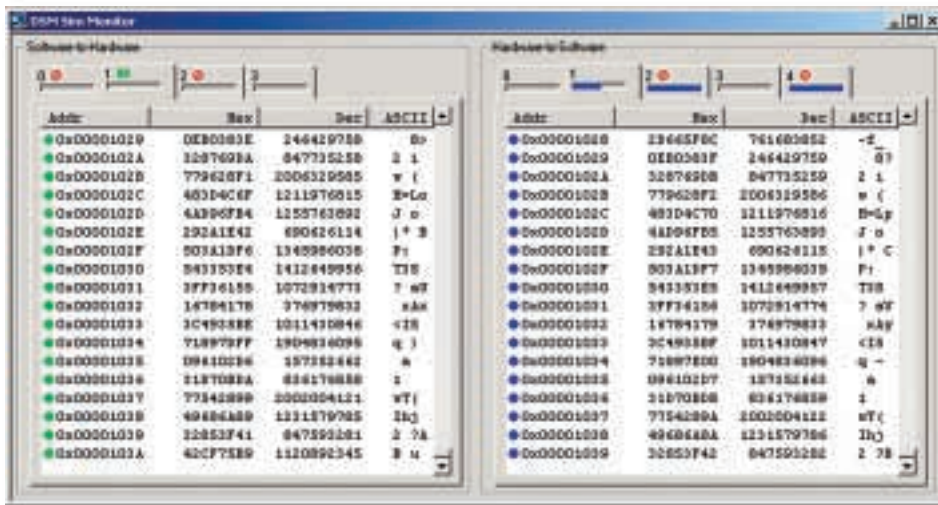


Figure 8 - DSM Sim Monitor for assisted debugging

	Encryption		Decryption	
	Software	FPGA	Software	FPGA
Elapsed time for 1 Mb of data	5558.8 ms	424.8 ms	5562.9 ms	424.7 ms
Cryptography rate	1.51 Mbps	19.7 Mbps	1.51 Mbps	19.8 Mbps

Table 1 - Performance statistics for triple DES encryption/decryption

tion/decryption, was programmed to generate VGA signals. Output from both the hardware and software implementations was merged to form a composite image on the monitor.

### Performance Comparison

A test harness enabled triple DES performance to be benchmarked by streaming data into either the software or hardware encryption algorithm.

Theoretical performance for the FPGA was calculated as follows: The triple DES implementation produced a 64-bit word every 19 clock cycles, giving a data throughput of 85.6 Mbps for a device running at 25.4 MHz.

Actual performance was profiled using WindView, a diagnostic tool from Wind River that enables visualization and analysis of performance and timing issues in embedded systems. It allowed triggers to be set at different points in the code, and then provided accurate timing information for each trigger event. Performance statistics are detailed in Table 1.

Platform	Clock Speed	Performance
SBC405	300 MHz	1.51 Mbps
Xilinx FPGA	20 MHz	33.8 Mbps

Table 2 - Performance comparison of hardware and software encryption

This test scenario showed an FPGA throughput about 13 times faster using hardware than running software on a PowerPC microprocessor. Nevertheless, it was still about a quarter of the theoretical maximum rate. This indicated that the full benefits of placing core routines in hardware might be compromised by other system bottlenecks. Further analysis showed that there were overheads associated with offloading functionality into hardware. These overheads were associated with RAM access latency and/or bus speeds.

We also calculated the performance of hardware and software encryption in the cartoon demonstration. Results demonstrated that the hardware performed 22 times faster on a 15 times slower clock, as shown in Table 2.

Following more detailed partitioning analysis, performance closer to the theoretical limits might be realized by removing code and functionality that are not directly associated with the triple DES algorithm (for example, the FLI decoder, frame buffer, and VGA driver). Better performance would also be achievable by connecting the FPGA directly to the processor bus in a memory-mapped fashion rather than across the PCI bus.

### Conclusion

The performance analysis results demonstrated significant improvements in overall system performance and quality of design. The results were achieved using a software-compiled system design methodology – specifically developed for programmable systems – that consistently delivered the fastest time to market (some 50% to 75% advantage in design time) without compromising performance or area.

For example, using the selected development tools and run-time environment, the FLI player took 10 person-days to implement, as did the triple-DES functionality. On the other hand, integrating these two blocks to produce the cartoon demonstration took just half a day. Moreover, you can very quickly explore the design space, experiment, and analyze different hardware/software trade-offs, and rapidly implement and prototype the system.

Coupling Celoxica's co-design technology with high-performance profiling tools in the development tool chain enabled further performance boosts and time-to-market efficiencies. Overall improvements in the quality of design were realized by more informed and accurate partitioning decisions, better up-front system verification, and by maximizing the speed gains of hardware implementation while minimizing the negative impact of transferring data between the FPGA and microprocessor.

The bottom line is that these system-level design qualities offer real and competitive advantages for designers of programmable systems who want to move to volume production. **Σ**